

Unit-4: Angular Features Angular Core Features

What is Angular?

Angular is a Type Script-based framework used to develop dynamic, single-page web applications. It provides a structured way to create user interfaces, manage data, handle user actions, validate forms, and communicate with web servers.

Features of Angular Framework

1. Component-based Architecture

- Angular applications are built using components as the primary building blocks
- Each component encapsulates its own template, logic, and styles
- Components are organized in a hierarchical tree structure
- Better reusability across the application, Easier to understand and maintain

2. Declarative Templates

- Declarative approach means you describe what should appear, not how to create it
- Templates use directives, bindings, and expressions
- Angular automatically updates the DOM when data changes
- Supports inline templates or external HTML files

3. Data Binding

- Data binding synchronizes data between the component class and the template
- Eliminates the need for manual DOM manipulation
- Angular automatically updates the view when model changes

4. Directives

- Directives are classes that add behavior to elements in Angular applications
- They extend HTML with custom functionality
- Three categories: Components, Structural Directives, and Attribute Directives

5. Pipes (Filters)

- Pipes transform data for display in templates, not modify the original data in the component
- Can be chained together for multiple transformations
- Angular provides built-in pipes and supports custom pipes

6. Forms and Validation

- Forms are essential for collecting user input, Support built-in validation.
- Angular provides two approaches: Template-driven and Reactive

7. Services and Dependency Injection

- Services contain reusable business logic and data operations.
- Keep components lean and focused on the view
- Dependency Injection (DI) is Angular's design pattern for providing dependencies

8. Routing

- Angular Router enables navigation between different views/components
- Implements single-page application (SPA) behavior
- URL-based navigation without full page reloads

9. HTTP Communication

- Angular's HttpClient module enables communication with backend servers
- Makes RESTful API calls
- Returns RxJS Observables for handling asynchronous operations
- Built-in support for request/response interceptors

Angular Components:

Components are the main building block of an angular application. Each component controls a specific part of the screen called a **view**. **Components** create UI using template, style sheet and class. The interactive UI are build using components. Each component has some data. Even components can share data to each other. All components are linked to *index.html* file.

Each **Components** includes

- **Template (HTML)** → defines the view.
- **Styles (CSS)** → defines the look.
- **Class (TypeScript)** → defines the behavior.

By-default Angular use **app. component** in *app* folder inside *src*.

Creating Components with @Component

The @Component decorator tells Angular that a class is a component and provides metadata about how it should be used.

Example:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-student',
  template: `<h2>{{student.name}}</h2>`,
  styles: [`h2 { color: blue; }`]
})
export class StudentComponent {
  student = { id: 1, name: "Pooja" };
}
```

- **Selector** → custom HTML tag (<app-student>) used in templates.
- **Template** → inline HTML or external file (templateUrl).
- **Styles** → inline CSS or external file (styleUrls).
- **Class** → TypeScript logic (properties, methods).

Component Structure Diagram

Angular Component

```
├── Selector (HTML Tag)
├── Template (HTML View)
├── Styles (CSS)
└── Class (TypeScript Logic)
```

Using the Component(app.component.html)

```
<h1>Welcome to Angular!</h1>
<app-student></app-student>
```

Advantages of Components

- Reusability, Easier maintenance
- Separation of responsibilities
- Better testing, Organized application structure
- Independent development of application sections

Component Lifecycle Stages

The component lifecycle in Angular consists of several stages:

- **Creation:** The component is instantiated and its dependencies are injected.
- **Change Detection:** Angular checks for changes in the data-bound properties.
- **Rendering:** The component's template is rendered or updated.
- **Destruction:** The component is destroyed and cleaned up.

Life cycle Hooks

Angular provides a set of lifecycle hooks that allow developers to execute code at specific stages of a component's lifecycle.

1. ngOnChanges

- It is called before ngOnInit (if the component has bound inputs) and whenever one or more data-bound input properties change. It is used to respond to changes in input properties.

```
ngOnChanges(changes: SimpleChanges) {  
    console.log('Changes detected:', changes);  
}
```

2. ngOnInit

- It is Called once, after the first ngOnChanges. It is used to initialize the component after Angular first displays the data-bound properties.

```
ngOnInit() {  
    console.log('Component initialized');  
}
```

3. ngDoCheck

- It is called during every change detection run, immediately after ngOnChanges and ngOnInit. It is used to detect and act upon changes that Angular can't or won't detect on its own.

```
ngDoCheck() {  
    console.log('Custom change detection');  
}
```

4. ngAfterContentInit

- It is called once after the first ngDoCheck. It is used to perform any additional initialization required for the content.

```
ngAfterContentInit() {  
    console.log('Content initialized');  
}
```

5. ngAfterContentChecked

- It is called after ngAfterContentInit and every subsequent ngDoCheck. It is used to act upon any changes after the content has been checked.

```
ngAfterContentChecked() {  
    console.log('Content checked');  
}
```

6. ngAfterViewInit

- It is called once after the first ngAfterContentChecked. It is used to perform additional initialization required for the view.

```
ngAfterViewInit() {  
    console.log('View initialized');  
}
```

7. ngAfterViewChecked

- It is called after ngAfterViewInit and every subsequent ngAfterContentChecked. It is used to act upon any changes after the view has been checked.

```
ngAfterViewChecked() {  
    console.log('View checked');  
}
```

8. ngOnDestroy

- It is called immediately before Angular destroys the component. It is used to clean up any resources, such as subscriptions and event handlers, to avoid memory leaks.

```
ngOnDestroy() {  
    console.log('Component destroyed');  
}
```

4.2 Component Communication

Components often need to exchange information. In a parent-child relationship:

- The parent sends data to the child using @Input().
- The child sends events to the parent using @Output() and EventEmitter.

Input, Output, and EventEmitter

Parent-to-Child Communication

The @Input() decorator allows a child component to receive data from its parent.

Theory of @Input()

- @Input() marks a property as a property that can receive external data.
- Data is passed using property binding.
- The parent controls the value.
- The child uses the received value.

Example: Passing Data Using @Input()***Child Component***

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <h3>Child Component</h3>
    <p>Student Name: {{ studentName }}</p>
    <p>Course: {{ course }}</p>
  `
})
export class ChildComponent {
  @Input() studentName: string = '';
  @Input() course: string = '';
}

Parent Component Class
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html'
})
export class ParentComponent {
  parentStudentName: string = 'Rahul';
  parentCourse: string = 'Web Development';
}

Parent Component Template
<app-child
  [studentName]="parentStudentName"
  [course]="parentCourse">
</app-child>
```

Child-to-Parent Communication

A child component uses @Output() to send an event to its parent.

Theory of @Output()

- @Output() exposes a custom event.
- EventEmitter creates and sends the event.
- The parent listens to the event using event binding.
- The child should not directly modify the parent's data.

Example: Sending an Event Using EventEmitter***Child Component***

```
import { Component, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendMessage()">
      Send Message to Parent
    </button>
  `
})
```

```

})
export class ChildComponent {
  @Output() messageEvent = new EventEmitter<string>();
  sendMessage(): void {
    this.messageEvent.emit('Hello from the child component!');
  }
}

```

Parent Component Class

```

import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html'
})
export class ParentComponent {
  receivedMessage: string = '';
  receiveMessage(message: string): void {
    this.receivedMessage = message;
  }
}

```

Parent Component Template

```

<app-child
  (messageEvent)="receiveMessage($event)">
</app-child>
<p>Message received: {{ receivedMessage }}</p>

```

4.2.1 Angular Directives: Structural Directives

A **directive** is a class that adds behavior to an element or changes the appearance or structure of the DOM.

Angular directives are generally divided into:

1. Components
2. Structural directives
3. Attribute directives

Structural Directives

Structural directives change the structure of the DOM by:

- Adding elements
- Removing elements
- Repeating elements

The asterisk syntax, such as `*ngIf`, is shorthand for an Angular template transformation.

For example: `<p *ngIf="isVisible">Visible content</p>`

Angular internally treats this conceptually as a template:

```

<ng-template [ngIf]="isVisible">
  <p>Visible content</p>
</ng-template>

```

1. *ngIf

- *ngIf conditionally adds or removes an element from the DOM.
- If the condition is true, Angular creates the element. If it is false, Angular removes it.

Syntax

```
<p *ngIf="isLoggedIn"> Welcome, user! </p>
```

Example: *ngIf with else

Component Class

```
export class LoginComponent {  
  isLoggedIn: boolean = false;  
}
```

Template

```
<button (click)="isLoggedIn = !isLoggedIn"> Toggle Login </button>  
<p *ngIf="isLoggedIn; else loggedOutMessage"> Welcome! You are logged in. </p>  
<ng-template #loggedOutMessage>  
  <p>Please log in to continue.</p>  
</ng-template>
```

then and else Syntax

```
<div *ngIf="isLoggedIn; then loggedIn; else loggedOut"></div>  
<ng-template #loggedIn>  
  <p>User dashboard is displayed.</p>  
</ng-template>  
<ng-template #loggedOut>  
  <p>Login page is displayed.</p>  
</ng-template>
```

Uses of *ngIf

- Showing login and logout messages
- Displaying loading indicators
- Showing validation messages
- Hiding unavailable features
- Conditional access to content

2. *ngFor

- *ngFor repeats an HTML element for each item in a collection.
- It is commonly used to display:
 - Student lists
 - Product lists
 - Menu items
 - Table rows
 - Messages

Syntax: <li *ngFor="let student of students"> {{ student }}

Example with Index and Position Variables**Component Class**

```
export class StudentListComponent {
  students: string[] = ['Asha', 'Ravi', 'Meena', 'John'];
}
```

Template

```
<ul>
  <li
    *ngFor="
      let student of students;
      let i = index;
      let first = first;
      let last = last;
      let even = even;
      let odd = odd
    ">
    {{ i + 1 }}. {{ student }}
    <span *ngIf="first"> - First student</span>
    <span *ngIf="last"> - Last student</span>
    <span *ngIf="even"> - Even row</span>
    <span *ngIf="odd"> - Odd row</span>
  </li>
</ul>
```

Important ngFor Variables

Variable	Meaning
index	Position of the current item
first	True for the first item
last	True for the last item
even	True when the index is even
odd	True when the index is odd

3. *ngSwitch

- ngSwitch displays one option from multiple possibilities.
- It is similar to a switch statement in programming.

It uses:

- [ngSwitch]
- *ngSwitchCase
- *ngSwitchDefault

Example: Component Class

```
export class GradeComponent {
  grade: string = 'A';
}
```

Template

```
<select [(ngModel)]="grade">
  <option value="A">A</option>
  <option value="B">B</option>
  <option value="C">C</option>
  <option value="D">D</option></select>
```

```
<div [ngSwitch]="grade">
  <p *ngSwitchCase="'A'"> Excellent performance </p>
  <p *ngSwitchCase="'B'"> Very good performance </p>
  <p *ngSwitchCase="'C'"> Good performance </p>
  <p *ngSwitchDefault> Needs improvement </p>
</div>
```

4.2.2 Attribute Directives

An **attribute directive** changes the appearance or behavior of an existing element without adding or removing it from the DOM.

Difference Between Structural and Attribute Directives

Structural Directive	Attribute Directive
Changes DOM structure	Changes appearance or behavior
Adds or removes elements	Keeps the element in the DOM
Uses asterisk syntax	Usually does not use asterisk syntax
Examples: *ngIf, *ngFor	Examples: ngClass, ngStyle

1. ngClass: ngClass dynamically adds or removes CSS classes.

String Syntax: `<div [ngClass]=" 'highlight' "> Highlighted content </div>`

Array Syntax: `<div [ngClass]=" [ 'bold-text', 'large-text' ] "> Styled content </div>`

Object Syntax: The object syntax applies a class when its condition is true.

```
<p [ngClass]="{
  'active': isActive,
  'disabled': isDisabled
}">
  Account status
</p>
```

Example: Component Class

```
export class StatusComponent {
  isOnline: boolean = true;
  isPremiumUser: boolean = false;
}
```

Template

```
<p
  [ngClass]="{
    'online-status': isOnline,
    'offline-status': !isOnline,
    'premium-status': isPremiumUser
  }">
  User Status
</p>
CSS
```

```
.online-status {
  color: green;
}.offline-status {
  color: red;
}.premium-status {
  font-weight: bold;
  background-color: gold;
}
```

2. ngStyle

- ngStyle dynamically applies inline CSS styles.
- It uses an object in which:
 - The key is the style property.
 - The value is the dynamic style value.

Example: Component Class

```
export class StyleComponent {
  textColor: string = 'blue';
  fontSize: number = 20;
  isImportant: boolean = true;
}
```

Template

```
<p
  [ngStyle]="{
    'color': textColor,
    'font-size.px': fontSize,
    'font-weight': isImportant ? 'bold' : 'normal'
  }">
  This paragraph has dynamic styling.
</p>
```

Direct Property Style Binding

```
<p [style.color]="textColor"> Dynamic color </p>
<p [style.font-size.px]="fontSize"> Dynamic font size </p>
```

4.3 Custom Directives

4.3.1 Creating Custom Directives

Custom directives are developer-created directives used to add reusable behavior to HTML elements.

They are useful when the same behavior is needed in multiple places.

Examples include:

- Highlighting an element
- Changing color on hover
- Automatically focusing an input
- Restricting keyboard input
- Displaying tooltips
- Applying permissions

@Directive Decorator

The @Directive decorator identifies a class as a directive.

```
@Directive({
  selector: '[appHighlight]'
})
```

The selector is written in square brackets because it represents an attribute.

ElementRef

ElementRef provides access to the host DOM element.

```
constructor(private element: ElementRef) {}
```

Direct DOM manipulation should be used carefully, especially in applications that support server-side rendering.

Renderer2

- Renderer2 provides a safer and more platform-independent way to modify the DOM.
- It can:
 - Set styles
 - Add classes
 - Remove classes
 - Set properties
 - Listen for events

@HostListener

@HostListener listens for events on the element to which the directive is applied.

```
@HostListener('mouseenter')
onMouseEnter(): void {
  // Event logic
}
```

@HostBinding

@HostBinding binds a directive property to a property of the host element.

```
@HostBinding('style.color')
textColor: string = 'blue';
```

4.3.2 Angular Data Binding

Data binding is the process of connecting component data with the HTML template. It keeps the model and view synchronized.

Angular provides four important types of data binding:

1. Interpolation
2. Property binding
3. Event binding
4. Two-way binding

1. Interpolation

Interpolation displays component data in the template.

It is a one-way data-binding technique from the component class to the view.

Syntax: {{ expression }}

Example

Component Class

```
export class ProfileComponent {  
  name: string = 'Priya';  
  age: number = 21;  
}
```

Template

```
<h2>{{ name }}</h2>  
<p>Age: {{ age }}</p>  
<p>Next year age: {{ age + 1 }}</p>  
<p>Student: {{ name.toUpperCase() }}</p>
```

Interpolation is commonly used for:

- Displaying text
- Displaying property values
- Performing simple expressions
- Calling simple methods

2. Property Binding

Property binding binds a component property to a DOM property.

Syntax: [property]="expression"

Example:

```
export class PropertyComponent {  
  imageUrl: string = 'assets/angular-logo.png';  
  isDisabled: boolean = true;  
  buttonText: string = 'Submit';  
}  
<img [src]="imageUrl" alt="Angular Logo">  
  
<button [disabled]="isDisabled"> {{ buttonText }} </button>
```

Interpolation v/s Property Binding

Interpolation is mainly used for displaying text: <p>{{ title }}</p>

Property binding is used for DOM properties:

```
<img [src]="imageUrl">  
<button [disabled]="isDisabled"></button>
```

The following may work for text-like values:

```
<button disabled="{{ isDisabled }}"> Submit </button>
```

However, property binding is preferred when binding Boolean, object, or non-string values.

3. Event Binding

Event binding allows the template to listen for user or browser events and call a class method.

Syntax: (event)="handler()"

Common events include:

- click
- input
- keyup
- change
- submit
- mouseenter
- mouseleave

Example

Component Class

```
export class EventComponent {
  message: string = '';
  inputValue: string = '';

  showMessage(): void {
    this.message = 'Button clicked';
  }

  handleInput(event: Event): void {
    const input = event.target as HTMLInputElement;
    this.inputValue = input.value;
  }

  handleKeyup(event: KeyboardEvent): void {
    console.log('Key pressed:', event.key);
  }
}
```

Template

```
<button (click)="showMessage()"> Click Me </button>

<p>{{ message }}</p>

<input
  type="text"
  (input)="handleInput($event)"
  (keyup)="handleKeyup($event)">

<p>Input value: {{ inputValue }}</p>
```

The \$event Object

\$event contains information about the event.

For an input event: `<input (input)="handleInput($event)">`

The event object can provide:

- The event type
- The target element
- The entered value
- Keyboard information

4. Two-Way Binding with [(ngModel)]

Two-way binding allows data to flow:

- From the component to the input
- From the input back to the component

It combines property binding and event binding.

Banana-in-a-Box Syntax

`[(ngModel)]="property"`

The square brackets and parentheses together are informally called the **banana-in-a-box syntax**.

FormsModule Requirement

To use ngModel, import FormsModule.

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    FormsModule
  ]
})
export class AppModule {}
```

Example

Component Class

```
export class TwoWayComponent {
  username: string = '';
}
```

Template

```
<label>Username:</label>

<input
  type="text"
  [(ngModel)]="username">

<p>You entered: {{ username }}</p>
```

4.4 Angular Pipes

A **pipe** transforms data in an Angular template without changing the original value in the component.

Pipes are useful for:

- Formatting text
- Formatting dates
- Formatting numbers
- Filtering or extracting data
- Converting values for display

Pipe Syntax: `{{ value | pipeName }}`

Pipe with Parameters: `{{ value | pipeName: parameter }}`

Chaining Pipe: Several pipes can be applied in sequence.

`{{ name | lowercase | titlecase }}`

Pure and Impure Pipes

Pure Pipe

- Executes only when the input value changes.
- More efficient.
- The default type of pipe.

Impure Pipe

- Executes during every change-detection cycle.
- Useful when the input object or array may change internally.
- Can affect performance.

4.4.1 Built-in Pipes

Assume the component contains:

```
export class PipeDemoComponent {  
  message: string = 'angular features';  
  studentName: string = 'ravi kumar';  
  price: number = 1250.5;  
  today: Date = new Date();  
  completion: number = 0.75;  
  topics: string[] = [  
    'Components',  
    'Directives',  
    'Pipes',  
    'Services'  
  ];  
}
```

1. uppercase: Converts text to uppercase.

`<p>{{ message | uppercase }}</p>`

Output: ANGULAR FEATURES

2. lowercase: Converts text to lowercase.

```
<p>{{ studentName | lowercase }}</p>
```

3. titlecase: Converts the first letter of each word to uppercase.

```
<p>{{ studentName | titlecase }}</p>
```

Output: Ranbir Kapoor

4. currency: Formats a number as currency.

```
<p>{{ price | currency }}</p>
<p>{{ price | currency: 'INR' }}</p>
<p>{{ price | currency: 'USD': 'symbol': '1.2-2' }}</p>
```

The format string controls the minimum and maximum number of decimal digits.

5. date: Formats a date.

```
<p>{{ today | date }}</p>
<p>{{ today | date: 'short' }}</p>
<p>{{ today | date: 'medium' }}</p>
<p>{{ today | date: 'longDate' }}</p>
<p>{{ today | date: 'dd/MM/yyyy' }}</p>
```

6. percent: Formats a decimal value as a percentage.

```
<p>{{ completion | percent }}</p>
<p>{{ completion | percent: '1.0-2' }}</p>
```

7. slice: Extracts part of a string or array.

```
<p>{{ message | slice:0:7 }}</p>
<ul>
  <li *ngFor="let topic of topics | slice:0:2">
    {{ topic }}
  </li>
</ul>
```

Creating a Custom Pipe

A custom pipe is created when Angular's built-in pipes do not meet an application's requirements.

A custom pipe contains:

1. @Pipe decorator
2. Pipe name
3. Class implementing PipeTransform
4. transform() method

@Pipe Decorator

```
@Pipe({
  name: 'reverse'
})
```

The name is used in the template.

Pipe Transform

The Pipe Transform interface requires the transform() method.

Example: Reverse String Pipe

Pipe Class

```
import {
  Pipe,
  PipeTransform
} from '@angular/core';
@Pipe({
  name: 'reverse',
  pure: true
})
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    if (!value) {
      return '';
    }
    return value.split('').reverse().join('');
  }
}
```

Using the Pipe

```
<p>{{ 'Angular' | reverse }}</p>
```

Output: ralugnA

Example: Custom Power Pipe

```
import {
  Pipe,
  PipeTransform
} from '@angular/core';
@Pipe({
  name: 'power'
})
export class PowerPipe implements PipeTransform {
  transform(value: number, exponent: number = 2): number {
    return Math.pow(value, exponent);
  }
}
<p>{{ 3 | power:3 }}</p>
```

The result is 27.

4.5 Angular Forms

Forms are used to collect and validate user input.

Common uses include:

- Login forms
- Registration forms
- Search forms
- Feedback forms
- Student information forms

Angular supports two main form approaches:

- Template-driven forms
- Reactive forms

Template-Driven vs Reactive Forms

Template-Driven Forms	Reactive Forms
Form logic is mostly in the template	Form logic is mainly in the component class
Uses FormsModule	Uses ReactiveFormsModule
Uses ngModel	Uses FormControl and FormGroup
Easier for simple forms	Better for complex forms
Less programmatic control	More programmatic control
More difficult to unit test	Easier to unit test

Template-Driven Forms

Template-driven forms are created mainly by using Angular directives in the HTML template.

Important directives include:

- ngForm
- ngModel
- ngModelGroup

Template-driven forms:

- Use two-way data binding.
- Are simple to create.
- Are suitable for small forms.
- Require Forms Module.
- Perform much of the work automatically.

Required Module

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [
    FormsModule
  ]
})
export class AppModule {}
```

Complete Template-Driven Form

Component Class

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-registration',
  templateUrl: './registration.component.html'
```

```

  })
  export class RegistrationComponent {
    user = {
      name: '',
      email: '',
      password: ''
    };
    submitForm(form: any): void {
      if (form.valid) {
        console.log('Form submitted:', this.user);
      }
    }
  }
}

```

Template

```

<form
  #registrationForm="ngForm"
  (ngSubmit)="submitForm(registrationForm)"
  novalidate>
  <label for="name">Name:</label>

  <input
    id="name"
    type="text"
    name="name"
    [(ngModel)]="user.name"
    required
    minlength="3"
    #name="ngModel">

  <div
    *ngIf="name.invalid && (name.dirty || name.touched)">

    <small *ngIf="name.errors?.['required']"> Name is required.</small>

    <small *ngIf="name.errors?.['minlength']">
      Name must contain at least three characters.
    </small>
  </div>
  <label for="email">Email:</label>
  <input
    id="email"
    type="email"
    name="email"
    [(ngModel)]="user.email"
    required
    email
    #email="ngModel">
  <div
    *ngIf="email.invalid && (email.dirty || email.touched)">
    <small *ngIf="email.errors?.['required']">
      Email is required.
    </small>
    <small *ngIf="email.errors?.['email']">
      Enter a valid email address.
    </small>
  </div>

```

```

<label for="password">Password:</label>
<input
  id="password"
  type="password"
  name="password"
  [(ngModel)]="user.password"
  required
  minlength="6"
  #password="ngModel">
<div
  *ngIf="password.invalid && (password.dirty || password.touched)">
  <small *ngIf="password.errors?.['required']">
    Password is required.
  </small>
  <small *ngIf="password.errors?.['minlength']">
    Password must contain at least six characters.
  </small>
</div>
<br>
<button type="submit" [disabled]="registrationForm.invalid"> Register </button>
</form>

```

Reactive Forms

Reactive forms define the form model in the component class.

Important classes include:

- FormControl
- FormGroup
- FormBuilder
- Validators

Required Module

```

import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  imports: [
    ReactiveFormsModule
  ]
})
export class AppModule {}

```

Complete Reactive Form

Component Class

```

import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  Validators
} from '@angular/forms';
@Component({
  selector: 'app-reactive-registration',
  templateUrl: './reactive-registration.component.html'
})
export class ReactiveRegistrationComponent {

```

```

registrationForm: FormGroup;
constructor(private formBuilder: FormBuilder) {
  this.registrationForm = this.formBuilder.group({
    name: [
      '',
      [
        Validators.required,
        Validators.minLength(3)
      ]
    ],
    email: [
      '',
      [
        Validators.required,
        Validators.email
      ]
    ],
    password: [
      '',
      [
        Validators.required,
        Validators.minLength(6)
      ]
    ]
  });
}
get name() {
  return this.registrationForm.get('name');
}
get email() {
  return this.registrationForm.get('email');
}
get password() {
  return this.registrationForm.get('password');
}
submitForm(): void {
  if (this.registrationForm.valid) {
    console.log(
      'Submitted data:',
      this.registrationForm.value
    );
  } else {
    this.registrationForm.markAllAsTouched();
  }
}
}

```

Template

```

<form
  [formGroup]="registrationForm"
  (ngSubmit)="submitForm()"
  novalidate>
  <label for="name">Name:</label>
  <input
    id="name"
    type="text"
    formControlName="name">
  <div *ngIf="name?.invalid && name?.touched">
    <small *ngIf="name?.errors?.['required']">
      Name is required.    </small>

```

```

    <small *ngIf="name?.errors?.['minlength']">Name must contain at least three characters.
  </small>
</div>
<br>
<label for="email">Email:</label>
<input
  id="email"
  type="email"
  formControlName="email">
<div *ngIf="email?.invalid && email?.touched">
  <small *ngIf="email?.errors?.['required']"> Email is required. </small>
  <small *ngIf="email?.errors?.['email']"> Enter a valid email address. </small>
</div>
<br>
<label for="password">Password:</label>
<input
  id="password"
  type="password"
  formControlName="password">
<div *ngIf="password?.invalid && password?.touched">
  <small *ngIf="password?.errors?.['required']"> Password is required. </small>
  <small *ngIf="password?.errors?.['minlength']">
    Password must contain at least six characters.
  </small>
</div>
<br>
<button
  type="submit"
  [disabled]="registrationForm.invalid">
  Register
</button>
</form>

```

Form Validation

Validation checks whether user-entered data meets specified requirements.

Built-in Validators

Validator	Purpose
required	Ensures a value is entered
minLength	Requires a minimum number of characters
maxLength	Limits the number of characters
email	Checks email format
pattern	Checks a regular expression pattern
min	Checks the minimum numeric value
max	Checks the maximum numeric value

Displaying Errors

A good practice is to show an error only when the control is invalid and the user has interacted with it.

```
<div *ngIf="email.invalid && (email.dirty || email.touched)">
  <p *ngIf="email.errors?.['required']"> Email is required. </p>
  <p *ngIf="email.errors?.['email']"> Invalid email format. </p>
</div>
```

Custom Validator Example

A custom validator returns:

- null when the value is valid
- An error object when the value is invalid

```
import {
  AbstractControl,
  ValidationErrors
} from '@angular/forms';
export function noSpaceValidator(
  control: AbstractControl
): ValidationErrors | null {
  const value: string = control.value || '';
  if (value.includes(' ')) {
    return { containsSpace: true };
  }
  return null;
}
```

Using the validator:

```
this.form = this.formBuilder.group({
  username: [
    '',
    [
      Validators.required,
      noSpaceValidator
    ]
  ]
});
```

Displaying the error:

```
<p *ngIf="username?.errors?.['containsSpace']"> Spaces are not allowed. </p>
```

4.6 Angular Services and Dependency Injection

4.6.1 Creating and Injecting Services

An Angular **service** is a class used to hold reusable logic and data operations.

Services commonly handle:

- Business logic
- API communication
- Shared data
- Logging
- Authentication
- Calculations
- Configuration
- Data transformation

A service prevents repeated code in components.

@Injectable Decorator

The @Injectable decorator tells Angular that a class can participate in dependency injection.

```
@Injectable({  
  providedIn: 'root'  
})
```

providedIn: 'root'

- Registers the service with the root injector.
- Makes it available throughout the application.
- Usually creates one shared instance.
- Avoids the need to add the service manually to a provider list.

Dependency Injection

Dependency Injection, or DI, is a design pattern in which a class receives the objects it needs from an external source instead of creating those objects itself.

Without DI: `const service = new StudentService();`

With DI: `constructor(private studentService: StudentService) {}`

Benefits of Dependency Injection

- Loose coupling
- Reusability
- Easier testing
- Centralized configuration
- Better maintainability
- Simplified object creation

Example: Creating and Injecting a Service

Service

```
import { Injectable } from '@angular/core';  
export interface Student {  
  id: number;  
  name: string;  
  course: string;  
}  
@Injectable({  
  providedIn: 'root'  
})  
export class StudentService {  
  private students: Student[] = [  
    {  
      id: 1,  
      name: 'Anita',  
      course: 'Angular'  
    },  
    {  
      id: 2,
```

```

    name: 'Vikram',
    course: 'TypeScript'
  }
];
getStudents(): Student[] {
  return this.students;
}
getStudentById(id: number): Student | undefined {
  return this.students.find(
    student => student.id === id
  );
}
addStudent(student: Student): void {
  this.students.push(student);
}
}

```

Using HttpClient to Call REST APIs

HttpClient is an Angular service used to communicate with web servers and REST APIs.

It supports common HTTP methods:

Method	Purpose
GET	Retrieve data
POST	Create new data
PUT	Replace existing data
PATCH	Partially update data
DELETE	Remove data

HttpClient methods return Observables.

Importing HTTP Support

In a module-based application:

```

import { HttpClientModule } from '@angular/common/http';
@NgModule({
  imports: [
    HttpClientModule
  ]
})
export class AppModule {}

```

In newer standalone applications:

```

import { provideHttpClient } from '@angular/common/http';
bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient()
  ]
});

```

4.6.2 Observables with RxJS

An **Observable** represents a stream of data that can arrive over time.

A stream may produce:

- One value
- Many values
- Values at regular intervals
- An error
- A completion notification

Examples include:

- HTTP responses
- User input events
- Timers
- WebSocket messages
- Route parameters

RxJS

RxJS means **Reactive Extensions for JavaScript**.

It provides:

- Observables
- Operators
- Subscriptions
- Error handling
- Asynchronous data processing

Observable

An Observable is a producer of values.

```
const numbers$ = of(1, 2, 3);
```

The dollar sign in a variable name is a common naming convention indicating that the variable contains an Observable.

Observer

An observer receives notifications from an Observable.

It can define:

- next
- error
- complete

Subscription

A subscription represents the active connection between an Observable and an observer.

```
const subscription = observable.subscribe(...);
```

A subscription can be cancelled: `subscription.unsubscribe();`

Creating and Subscribing to an Observable

```
import { Observable } from 'rxjs';
const numbers$ = new Observable<number>((
  subscriber => {
    subscriber.next(10);
    subscriber.next(20);
    subscriber.next(30);
    subscriber.complete();
  }
));
numbers$.subscribe({
  next: value => {
    console.log('Value:', value);
  },
  error: error => {
    console.error('Error:', error);
  },
  complete: () => {
    console.log('Observable completed');
  }
});
```

Output

Value: 10

Value: 20

Value: 30

Observable completed

Handling HTTP Responses with Observables

```
import { Component, OnInit } from '@angular/core';
import { catchError, map, of } from 'rxjs';
@Component({
  selector: 'app-data',
  template: `
    <p *ngFor="let name of names">
      {{ name }}
    </p>
    <p *ngIf="errorMessage">
      {{ errorMessage }}
    </p>
  `
})
export class DataComponent implements OnInit {
  names: string[] = [];
  errorMessage: string = '';
  constructor(
    private studentService: StudentService
  ) {}
  ngOnInit(): void {
    this.studentService.getStudents()
      .pipe(
        map(students =>
          students.map(student => student.name)
        ),
        catchError(error => {
          this.errorMessage =
            'Could not load student data.';
        })
      )
      .subscribe();
  }
}
```

```

        return of([]);
    })
)
.subscribe(names => {
    this.names = names;
});
}
}

```

Unsubscribing from Observables

Some Observables continue emitting values for a long time. If a component is destroyed while the subscription remains active, it may cause:

- Memory leaks
- Unwanted background work
- Unexpected updates
- Errors after navigation

HTTP Observables usually complete automatically after one response. However, long-running Observables such as interval, valueChanges, and WebSocket streams may require cleanup.

Manual Unsubscription

```

import {
  Component,
  OnDestroy,
  OnInit
} from '@angular/core';
import {
  interval,
  Subscription
} from 'rxjs';
@Component({
  selector: 'app-timer',
  template: `
    <p>Seconds: {{ seconds }}</p>
  `
})
export class TimerComponent
  implements OnInit, OnDestroy {
  seconds: number = 0;
  timerSubscription?: Subscription;

  ngOnInit(): void {
    this.timerSubscription = interval(1000)
      .subscribe(value => {
        this.seconds = value;
      });
  }

  ngOnDestroy(): void {
    this.timerSubscription?.unsubscribe();
    console.log('Timer subscription removed');
  }
}

```

Recommended Angular Pattern: takeUntil

```

import {
  Component,

```

```
    OnDestroy,
    OnInit
  } from '@angular/core';
  import {
    Subject,
    interval
  } from 'rxjs';
  import {
    takeUntil
  } from 'rxjs/operators';
  @Component({
    selector: 'app-cleanup',
    template: `
      <p>Observable is active.</p>
    `
  })
  export class CleanupComponent
    implements OnInit, OnDestroy {
    private destroy$ = new Subject<void>();
    ngOnInit(): void {
      interval(1000)
        .pipe(
          takeUntil(this.destroy$)
        )
        .subscribe(value => {
          console.log(value);
        });
    }
    ngOnDestroy(): void {
      this.destroy$.next();
      this.destroy$.complete();
    }
  }
}
```